Certificación PCAP

Lección 4: Cadenas y Listas II

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Lección 4: Cadenas y listas II

1. Objetivos

En esta nueva lección continuaremos con el repaso y/o estudio del contenido del examen PCAP.

Concretamente, centraremos nuestra atención en la definición y estudio de las principales características de los datos de tipo “cadena” o “strings” y los principales métodos y funciones que Python pone a nuestro alcance para trabajar con ellos. Posteriormente, pasaremos a ver una segunda parte de los datos de tipo lista, centrando nuestra atención en la definición y manipulación de listas multidimensionales, así como en algunas operaciones y métodos importantes para trabajar con listas.

Nuevamente, repasamos los aspectos que pudieran tener más importancia de cara al examen, no pretendiendo en ningún caso ser la presente una guía completa ni de cadenas ni de listas multidimensionales, ya que dicho tema daría mucho más de sí para explicarse en detalle.

2. CADENAS (STRINGS)

Un tipo de datos muy importante en Python y en cualquier otro lenguaje, son las llamada cadenas o strings, aparecen en casi cualquier desarrollo y, como se podrá intuir, también son muy frecuentes en casi todas las preguntas del examen del PCAP. Es por ello que en este apartado centraremos nuestra atención en este tipo de datos donde repasaremos sus características principales y las funciones y métodos más utilizados para trabajar con ellos.

¿Qué tipo de datos son las cadenas?

Pues bien, las cadenas se definen como una **secuencia ordenada inmutable de longitud arbitraria pero finita de caracteres (letras, números y caracteres especiales).**

La manera en la que se definen las cadenas en Python es la siguiente:

cadena = “esto es una cadena”

Y en el siguiente ejemplo vemos una asignación real.

```
1.      >>> cadena = "abcABC123"
```

Vemos como para definir las cadenas es necesario encerrar entre comillas dobles los caracteres que queramos que nuestra variable contenga. Aun así, también es posible definir cadenas en Python encerrando la secuencia de caracteres entre dos apóstrofes .

cadena = ‘Esto también es una cadena’

Las cadenas no son las únicas que pueden ser consideradas como “secuencias ordenadas”, las listas y tuplas, por ejemplo, también lo son. Por lo tanto, compartirán con estas una serie de características que repasaremos a continuación.

2.1. Indexación

Una de las primeras propiedades que las cadenas comparten tanto con listas como con tuplas es la indexación. Esta indexación hace referencia a la posibilidad de recuperar/leer los valores relativos a un determinado “índice” que da cuenta de la “posición” en la que se encuentra un determinado dato, o en el caso de las cadenas, carácter.

Veamos un ejemplo de esta indexación

```
>>> cadena = "La indexación es posible"
>>> print(cadena[3])
i
```

Como podemos observar, la sintaxis de la indexación es exactamente igual que la de las listas y tuplas. Se utiliza el nombre de la variable seguida del índice del carácter que se desea recuperar entre corchetes.

cadena[índice]

Otro de los detalles que podemos observar es que, al primer carácter de la cadena, al igual que al primer elemento de las listas y tuplas, se le asigna el índice "0".

```
>>> cadena = "abcABC123"
>>> print(cadena[0])
a
```

Continuando con las similitudes, en las cadenas también es posible utilizar índices negativos. Se siguen las mismas normas que para el caso de las listas y tuplas.

El índice "-1" hace referencia al último carácter, el "-2" al penúltimo y así sucesivamente, y los límites para estos pueden resumirse haciendo uso de la función **len(),** la cual funciona igualmente en las cadenas, solo que en vez de devolver el número de elementos devuelve el número de caracteres totales de la cadena en cuestión.

-len(cadena) <= índice <= len(cadena)-1

Por lo tanto, utilizar un índice fuera de estos límites arrojará un error, ya que, en dicho caso, Python entenderá que se estará intentando recuperar un carácter inexistente.

```
>>> cadena = "abcABC123"
>>> len(cadena)
9
>>> print(cadena[-1], cadena[-2], sep = ",")
3,2
>>> print(cadena[10])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

2.2. Slices

La potencia y versatilidad de los slices también está a nuestro alcance al trabajar con cadenas. Por lo que todo lo visto en el caso de las listas es aplicable en este tipo de datos.

Pondremos algunos ejemplos a continuación.

```
>>> cadena
'abcABC123'
>>> print(cadena[1:6:2])
bAC
>>> print(cadena[6:])
123
>>> print(cadena[:4])
abcA
>>> print(cadena[1:6:2])
bAC
>>> print(cadena[-1::-1])
321CBAcba
>>> print(cadena[6:2])

```

Vemos como las propiedades de los slices son idénticas al trabajar con las cadenas.

Igualmente, se cree conveniente recalcar el resultado del siguiente tipo de slice:

```
>>> print(cadena[-9:8])
abcABC12

```

En **ningún caso** se mostrará por pantalla la cadena duplicada. Los índices no guardan relación entre sí, simplemente hacen referencia a posiciones de caracteres de la cadena en cuestión. Tanto con listas, tuplas o cadenas, en el examen hay altas probabilidades de que nos encontremos preguntas de este estilo, así que merece la pena tenerlo siempre en cuenta.

2.3. Inmutabilidad de las cadenas

Como se ha comentado en la definición de cadenas, uno de los aspectos más importantes en la misma es el hecho de que las cadenas son un tipo de dato **inmutable**, lo cual claramente quiere decir que intentar alterar su contenido no será posible, al menos de manera directa, es decir, sin necesidad de crear nuevos objetos que sean **copias** alteradas del objeto original. Lo veremos a continuación.

Lo importante es tener claro que la siguiente situación arrojará un error que nos advertirá claramente de la inmutabilidad de este tipo de datos. Veamos un ejemplo.

```
>>> cadena
'abcABC123'
>>> cadena[3] = "J"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item
assignment
```

Podemos leer claramente en el mensaje de error que este tipo de variables no soporta las modificaciones.

Pero, por otro lado, sí que podemos hacer lo siguiente.

```
>>> cadena
'abcABC123'
>>> cadena = cadena + "esto es una cadena añadida"
>>> cadena
'abcABC123esto es una cadena añadida'
```

¿Cómo puede ser esto posible si acabamos de recalcar que las cadenas en Python son tipos de datos inmutables?

Pues bien, "la historia completa" de este último ejemplo es la siguiente.

Cuando hacemos cadena= "abcABC123" Python crea el **objeto de tipo cadena** "abcABC123" y asigna el nombre de variable "cadena" a este objeto para poder apuntar a él en el momento que se requiera.

Posteriormente, cuando hacemos cadena = cadena + “esto es una cadena añadida” en la línea 3, lo que Python entiende es: puesto que “cadena” es un objeto de tipo inmutable, **creemos un nuevo objeto** de tipo cadena con los siguientes caracteres “abcABC123esto es una cadena añadida” y asignemos el nombre “cadena” a este nuevo objeto (dejando el antiguo de lado).

Lo que nosotros percibimos por pantalla es que el objeto se ha modificado, pero lo que en realidad ocurre es que se crea un nuevo objeto y se reutiliza el nombre de la variable dejando “huérfano” al antiguo objeto “abcABC123”.

No nos quedemos solo en las palabras, comprobemos esta manera de proceder que tiene Python con los objetos mutables vs los objetos inmutables. Para ello haremos uso del mismo ejemplo que utilizamos en la mutabilidad de las listas.

Es decir, asignaremos el mismo objeto a dos nombres de variables distintos y posteriormente intentaremos mutar una de las variables e imprimiremos el resultado.

Ejemplo para listas (mutables)

```
1.      >>> lista1 = [1,2,3,4]
2.      >>> lista2 = lista1 # Hacemos que las dos variables
3.                           # apunten al mismo objeto
4.      >>> lista2 += ["nuevo elemento"] #Intentamos
5.                           # modificar el objeto.
6.      >>> print(lista1, lista2, sep=" || ")
7.      [1, 2, 3, 4, 'nuevo elemento] || [1, 2, 3, 4,
8.      'nuevo elemento']
9.      #Como el objeto era mutable, no se crea uno nuevo
10.
```

Podemos ver que al imprimir las dos variables en la línea 5, los resultados obtenidos son iguales, es decir, al haber alterado la variable **lista2** hemos alterado la variable **lista1**. Esto significa que, puesto que el objeto inicial **lista1 = [1,2,3,4]** al que tanto los nombres “lista1” y “lista2” apuntaban, era de tipo mutable, no se creará uno nuevo al intentar modificarlo, y una vez realizada la modificación, ambos nombres seguirán apuntando al mismo objeto. Esta es la explicación de por qué tras haber alterado **lista2** observamos los cambios imprimiendo **lista1**.

Ejemplo para cadenas (inmutables)

```
>>> cadena1 = "abcdefg"
>>> cadena2 = cadena1 # Hacemos que las dos variables
                      # apunten al mismo objeto
>>> cadena2 += "carac añadidos" #Intentamos
                      # modificar el objeto.
>>> print(cadena1, cadena2, sep=" || ")
abcdefg || abcdefgcarac añadidos
#Como el objeto no era mutable, se crea uno nuevo

```

En este ejemplo utilizamos el mismo proceder que en el caso de las listas. Asignamos dos nombres: **cadena1** y **cadena2** al mismo objeto de tipo cadena inmutable. Posteriormente, modificamos **cadena2** e imprimimos las dos variables. Vemos que ahora los resultados difieren. Es decir, se crea un nuevo objeto y se asigna a la variable **cadena2**.

Otra de las consecuencias de que las cadenas sean objetos inmutables en Python es que no podremos utilizar la instrucción **del** para borrar ninguno de sus caracteres. Sin embargo, esta instrucción sí podremos utilizarla para borrar la cadena completa.

```
>>> cadena = "abcABC123"

>>> del cadena[3] # Intentamos borrar el cuarto
                  # caracter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item deletion

>>> del cadena # Borramos la cadena completa
>>> cadena     # Intentamos acceder a la variable
               # pero ésta ya no existe.
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'cadena' is not defined

```

Pese a sus muchas similitudes, las cadenas y las listas son objetos de distintas clases en Python y consecuentemente, los métodos a los que cada una de ellas tendrá acceso también lo serán. Es el caso, por ejemplo, del método **append()**, dónde, además de ser un método perteneciente a otra clase, sirve para "mutar" las variables lo cual no está permitido en las cadenas en Python.

2.4. Algunas operaciones útiles con cadenas

Podríamos estar tentados de pensar que dado que las cadenas son inmutables puede resultar complejo trabajar con ellas y darles utilidad. Nada más lejos de la realidad, las cadenas son ampliamente utilizadas en diversidad de situaciones y su inmutabilidad es simplemente una cualidad que tenemos que tener en cuenta cuando trabajemos con ellas para evitar cometer errores.

A continuación, veremos una serie de funciones, métodos y operadores que pueden resultar muy útiles para interactuar con objetos de tipo cadena y que nuevamente, son recurrentes en el examen del PCAP.

2.4.1. Concatenación y replicación

En la lección 2 hablamos extensamente sobre operadores aritméticos y sus características, sin embargo, como es lógico sólo los utilizamos con objetos de tipo entero o flotante, ya que las operaciones matemáticas que representan tienen sentido solo con este tipo de datos.

Aun así, algunos de ellos (concretamente el **+** y el *****) es posible utilizarlos también con datos de tipo cadena, donde el comportamiento que tendrán será un tanto distinto, aunque en cierta manera también lógico.

Cuando utilizamos el operador con símbolo **+** entre cadenas, la operación realizada ya no se dice que es una suma aritmética, sino que se denomina **concatenación**, y como su nombre indica, el resultado devuelto contiene las dos cadenas unidas en el orden en el que se invocó dicha concatenación.

```
1.      >>> cadena1 = "hola mundo"
2.      >>> cadena2 = " esto es una concatenación"
3.      >>> print(cadena1 + cadena2)
4.      hola mundo esto es una concatenación
5.
6.      >>> print(cadena2 + cadena1)
7.      esto es una concatenaciónhola mundo
8.
```

El operador de concatenación abreviada "**+=**" también es válido y su funcionamiento es análogo al caso de la suma aritmética, solo que con la concatenación.

"cadena1" += "cadena2"
"cadena1" = "cadena1" + "cadena2"

Tabla 2.1. Equivalencia del operador de concatenación abreviada con el operador concatenación

Las expresiones anteriores son equivalentes, podemos verlo en el siguiente ejemplo.

```
>>> cadena1 = "Hola mundo"
>>> cadena1 += " esto es una concatenación"
>>> print(cadena1)
Hola mundo esto es una concatenación

```

Del mismo modo, podemos hacer uso del operador con símbolo " ***** ". Al utilizarlo con números, la operación recibe el nombre de multiplicación, sin embargo, al tratar con cadenas se denomina replicación.

Su función es distinta pero bastante intuitiva, ya que como podemos deducir por su nombre, al invocarlo, Python construirá un nuevo objeto de tipo cadena cuyo contenido será el resultado de replicar N veces la cadena inicial.

Por lo tanto, para funcionar, la operación de replicación necesita que uno de los operandos sea un objeto de tipo **cadena** y el otro un **entero**, si utilizáramos un flotante, obtendríamos un fallo.

Al igual que con la concatenación, podemos utilizar el operador abreviado "***=**" obteniendo el resultado esperado. Veámoslo a continuación.

```
>>> cadena = "ABCDEF"
>>> print(cadena * 2)
ABCDEFABCDEF

>>> cadena *= 2
>>> print(cadena)
ABCDEFABCDEF

>>> print(cadena * 2.3)
Traceback (most recent call last):
  TypeError can't multiply sequence by non-int of
type 'float'
```

2.4.2. Funciones ord() y chr()

Estas funciones pueden resultar muy interesantes tanto desde el punto de vista del examen como útiles para poder hablar posteriormente de la comparación entre cadenas.

La función **ord()** necesita como argumento de **entrada una cadena de longitud 1**, es decir, un carácter, y su resultado es su correspondiente código de la tabla ASCII/Unicode.

Importante: *introducir una cadena de longitud mayor arrojará un error.*

```
>>> print(ord("A"), ord("a"),sep = ", ")
65, 97
>>> print(ord("AB"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ord() expected a character, but string of
 length 2 found

```

Por otro lado, tenemos la función inversa: la función **chr()**. Esta función recibe como argumento de entrada el código (número entero) de la tabla ASCII/unicode y su resultado es el carácter asociado.

```
>>> print(chr(65), chr(97), sep = ", ")
A, a

>>> print(chr(1.2))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: integer argument expected, got float

>>> print(chr(99999999))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: chr() arg not in range(0x110000)

```

Como podremos imaginar, esta función no soporta valores que no sean enteros positivos y puesto que la tabla de códigos es finita, también tiene un entero máximo a partir del cual la función devolverá un error tal y como podemos observar en el ejemplo anterior.

Las funciones ord() y chr() son inversas, es decir, aplicar ord() a una cadena y a su resultado aplicar chr(), devolverá la cadena inicial.

chr(ord("cadena")) = cadena

Respecto a estas funciones y el trabajo con cadenas, está claro que no se nos preguntará en el examen los códigos de los caracteres de memoria, pero sí que es importante conocer el hecho de **que los caracteres mayúscula tienen códigos inferiores a los caracteres minúscula**, ya que en las operaciones de comparación de caracteres que veremos a continuación sí que pueden aparecer en el examen.

2.4.3. Comparación de cadenas

Pese a que pueda parecer un tanto extraño, Python ofrece la posibilidad de comparar cadenas utilizando los operadores "**==, !=, <, >, <=, >=**". Para ello, Python hará uso de los códigos de la tabla ASCII/unicode.

Los resultados que se pueden obtener al realizar este tipo de comparaciones pueden resultar un poco sorprendentes, por ello a continuación iremos viendo algunas de las casuísticas más relevantes y que tenemos alguna posibilidad de encontrar en el examen.

> Dos cadenas son iguales cuando están formadas por los mismos caracteres en el mismo orden.

```
>>> print("abc" == "abc")
True
>>> print("abc" == "abcd")
False
```

> Tanto si las cadenas son de la misma longitud como si no, Python utilizará el primer carácter que sea diferente para calcular la expresión. Utilizará los códigos de esos caracteres para realizar la comparación (ya hemos visto que las letras mayúsculas tienen códigos inferiores).

```
>>> print("abA" > "aba")
False
>>> print("abA" < "aba")
True
>>> print("abc" < "abcd")
True
>>> print("abcdefg" < "dbcd")
True
```

Hemos marcado el primer carácter no coincidente en los ejemplos anteriores y, cómo podemos ver en todos los casos, ese es el carácter clave para determinar el resultado de la comparación.

Otra posible situación es que las cadenas estén formadas por dígitos. Veamos algunos ejemplos

```
>>> print("1" > "0")
True
>>> print("1" > "00")
True
>>> print("1" > "20")
False
>>> print("2" > "120")
True
```

Debemos tener claro que la cadena “10” no es equivalente al entero 10. Python continúa utilizando las mismas normas para comprar cadenas, ya que, al fin y al cabo, es lo que son. Compara carácter a carácter y utiliza la primera diferencia para realizar la comparación y determinar el resultado de la misma.

Podemos darle una vuelta de tuerca más para complicar la situación. Podemos intentar comparar cadenas que contengan dígitos, como por ejemplo “10” con enteros, como el 10. Esta práctica no resulta muy recomendable y solo es posible realizarla sin que Python arroje un error con los operadores “**==**” y “**!=**”. En el primer caso obtendremos siempre “False” mientras que en el segundo obtendremos siempre “True”. Intentar utilizar otros operadores (<,>,<=,>=) devolverá un error donde Python nos dirá que no soporta ese tipo de comparación entre tipos de dato *string* y *entero (*o float).

```
>>> "10" == 10
False
>>> "10" != 10
True
>>> "10" > 9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances
of 'str' and 'int'
>>> "10" > 9.8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances
of 'str' and 'float'
```

Para realizar este tipo de comparaciones de una manera segura, Python pone a nuestra disposición la posibilidad de "transformar" los tipos de datos mediante las funciones **str(), int()** y **float()**.

> **Funcion str()**. Esta función se encarga de convertir los números, ya sean enteros o flotantes en strings. Siempre será posible la transformación en este sentido.

```
>>> str(12)
'12'
>>> str(1.3)
'1.3'
```

> **Funcion int()**. Como podremos imaginar por su nombre, esta función se encarga de transformar tanto una **cadena** como un **flotante** a números enteros. En este caso, sí que hay que tener cuidado al usarla, porque esta transformación no siempre es posible, ya que cualquier número puede transformarse en una cadena (solamente hay que ponerlo entre comillas) pero en el sentido contrario esto no es así, ya que no todas las cadenas pueden tener sentido como números ("a" nunca puede entenderse como un número mientras que "12" es fácilmente convertible). De la misma manera, debemos tener en cuenta que al aplicarse esta función sobre un flotante, siempre obtendremos un entero, la única particularidad es que el entero resultante siempre será truncado y no redondeado.

```
>>> int(3.2)
3
>>> int(3.9) # Al realizar la transformación se
             # trunca
3
>>> int("15")
15
>>> int("cadena")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with
base 10: 'cadena'
```

Función float(). Esta función intenta convertir en flotantes tanto enteros (los que no presentarán ningún tipo de problema a la hora de trabajar con ellos) como con cadenas, donde nuevamente tendremos que ser cuidadosos, ya que mientras que "4.23" es fácil ver que podrá ser transformado sin problema, "abc" arrojará un error.

```
>>> float(2)
2.0
>>> float(2.2)
2.2
>>> float("4")
4.0
>>> float("4.4")
4.4
>>> float("4,4")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float: '4,4'
```

Igualmente, debemos ser cautelosos con el uso de la coma "," en lugar del punto"." para la separación de los decimales. Python no será capaz de realizar la transformación a flotante si utilizamos este símbolo. Podemos ver como al utilizarlo arroja un error.

3. Listas II

En esta sección veremos algunas propiedades extra de las variables de tipo lista, ya que además de ser estas muy importantes en la programación en general, las preguntas del examen en cuyo contenido podemos encontrarlas son numerosas.

3.1. Listas Multidimensionales: definición y principales características

Como ya vimos en las lecciones anteriores, la definición de una lista podía hacerse de la siguiente manera.

lista = [elemento1, elemento2, elemento3,...,elementoN]

Vimos además que las listas podían estar compuestas por cualquier cantidad de elementos y que, además, estos no tenían por qué ser todos de la misma clase, ya que podíamos mezclar cadenas, enteros, flotantes e incluso otras listas. Es aquí donde queremos centrar nuestra atención en este apartado, como construir listas multidimensionales o lo que es lo mismo listas cuyos elementos sean listas y que a su vez estos puedan contener otras y así sucesivamente.

Las listas multidimensionales, por lo tanto, pueden construirse de la siguiente manera.

lista = [[el_00, el_01, el_02,...el_0N], [el_10, el_11, el_12,...el_1N],...[el_N0, el_M1, el_M2,...el_MN]]

Llevado a un ejemplo de lista sencilla:

```
1.      >>> lista = [[1,2,3], [4,5,6], [7,8,9]]
2.      >>> print(lista)
3.      [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
4.
```

Este tipo de lista puede resultarnos familiar de un concepto matemático; sí, las matrices. Básicamente, al hablar de listas multidimensionales, estamos hablando de matrices (no en el sentido estricto).

Al definir la lista del ejemplo anterior podemos verlo como la siguiente matriz.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Igualmente, como en todas las listas, es posible indexar cada uno de los elementos y aplicar todo lo visto en la primera parte de las listas a estas listas multidimensionales. De tal manera que si queremos recuperar algún elemento que se encuentre en la fila "i" y columna "j" de ella no tenemos más que recuperarla como vemos a continuación

elemento_ij = lista[i][j]

```
1.      >>> lista[1][2]
2.      6
3.      >>> lista[0][2]
4.      3
5.      >>> lista[1][2]
6.      6
7.      >>> lista[1][3]
8.      Traceback (most recent call last):
9.        File "<stdin>", line 1, in <module>
10.     IndexError: list index out of range
11.
```

El hecho de que las listas puedan ser de dimensión 2 o dimensión N no altera las propiedades y características que repasamos en lecciones previas. Por lo tanto, no debe extrañarnos el error que vemos en la línea 8 del ejemplo anterior, donde hemos intentado acceder a un elemento inexistente y consecuentemente Python nos ha avisado que el índice introducido está fuera del rango de elementos existentes.

Mediante esta indexación es posible también alterar el valor de cualquier elemento de la lista, siempre que conozcamos sus índices. Bastará hacerlo como sigue.

Lista[i][j] = nuevo_valor

```
1.      >>> lista
2.      [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
3.      >>> lista[0][2] = "elemento alterado"
4.      >>> print(lista)
5.      [[1, 2, 'elemento alterado'], [4, 5, 6], [7, 8, 9]]
6.
```

Los slices son igualmente válidos y útiles para trabajar con listas multidimensionales. Puede resultar un poco más complejo tratar con más de una dimensión desde nuestro punto de vista, pero las propiedades vistas para listas mono dimensionales siguen aplicándose en este caso.

Veamos estos slices en acción en unos pocos ejemplos.

```
>>> lista
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> lista[0:][0:]
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> lista[0:-1][0:1]
[[1, 2, 3]]

>>> lista[:2][:2]
[[1, 2, 3], [4, 5, 6]]

>>> lista[-1::-1][1]
[4, 5, 6]

>>> lista[-1::-1][:]
[[7, 8, 9], [4, 5, 6], [1, 2, 3]]

```

Imaginemos ahora que necesitamos crear una lista de dimensiones 100 x 100, sería muy tedioso realizar la definición de la lista manualmente ¿verdad? Es por ello que a continuación veremos cómo simplificar este proceso mediante el uso de bucles "**for"** y de las llamadas **"compresiones**".

3.2. Bucle "for" para crear listas

El bucle "for" puede resultarnos de mucha ayuda a la hora de definir listas de dimensiones y cantidad de elementos demasiado grandes como para definirlas a mano.

Supongamos primeramente que nos interesa crear una lista mono dimensional con todos sus elementos puestos a cero pero de longitud 30. Podríamos perfectamente hacerlo a mano, pero utilizando el bucle "for" sería mucho más inmediato.

```
>>> lista = []
>>> for i in range(31):
>>>     lista.append(0)
>>> print(lista)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> print("La longitud de la lista es: ",len(lista))
La longitud de la lista es:  30

```

Ahora, bien, si quisiéramos una lista de dimensión 2 con la misma longitud de elementos, la tarea manualmente comenzaría a ser un tanto inabarcable, mucho menos si hablásemos de longitudes mayores. Por lo tanto, en este caso sí o sí tendríamos que hacer uso del bucle "for". Podríamos definir una lista de 10 x 10 elementos de la manera siguiente.

```
>>> lista = []
>>> for i in range(10):
>>>     fila = []
>>>     for j in range(10):
>>>         fila.append(0)
>>>     lista.append(fila)

>>> for fila in lista:
>>>    print(fila)

 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

```

Aun así, podríamos simplificar un poco más la creación de esta lista, veamos cómo.

```
>>> lista = []
>>> for i in range(10):
>>>    lista.append([0] * 10)
>>> for fila in lista:
>>>     print(fila)

```

Vemos que, en este segundo ejemplo, hemos sido capaces de reducir las líneas de código necesarias para realizar la misma tarea. Sin embargo, ¿será posible simplificar más aun este procedimiento?

La respuesta es que sí, y no ha de sorprendernos, ya que entre las características principales de Python se encuentra la posibilidad de escribir código de la manera más corta, clara y limpia posibles, y este caso no es una excepción.

Para llevar a cabo esta simplificación "máxima" de la definición de una lista, haremos uso de las llamadas **compresiones**. Su nombre es bastante acertado, ya que mediante su uso estaremos comprimiendo la cantidad de código necesario para realizar la definición de las listas.

3.2.1. Compresión de listas

Esta manera de definir listas es muy útil al trabajar con listas multidimensionales, no significa esto que no sea utilizada en listas de dimensión 1, simplemente es cuando tratamos con varias dimensiones cuando es más evidente la cantidad de tiempo y esfuerzo que nos ahorra su utilización.

Veamos la sintaxis que utilizamos para definir en primer lugar una lista mono dimensional.

lista = [“*expresión*” for “*elemento*” in “secuencia_iterable”]

Se trata básicamente de la utilización de un bucle "for" en una misma línea para definir la lista.

Para definir una lista mediante una compresión, es importante que utilicemos los corchetes "[]" ya que de esta manera le estaremos diciendo a Python que el tipo de dato que contendrá la variable "Lista" será de tipo lista, valga la redundancia. Las palabras clave que tenemos que utilizar son **for** e **in**, las mismas que en el bucle "for" por supuesto.

Finalmente, es igualmente importante la utilización de una **secuencia iterable**, un variable **elemento** que represente a cada uno de los elementos de dicha secuencia, y también la **expresión** que determinará el valor de cada uno de los elementos que contendrá la lista.

Veamos un ejemplo de definición de lista mediante las compresiones.

```
>>> lista = [i for i in range(10)]
>>> print(lista)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Podemos observar como la "secuencia iterable" está proporcionada por la función range(). En la mayoría de los ejemplos veremos su utilización, sin embargo, las compresiones no solo funcionan con ella, sino que cualquier secuencia iterable será válida. Es posible utilizar otras como listas, tuplas o cadenas, ya que todas ellas son iterables. Lo vemos en los siguientes ejemplos.

```
>>> lista1 = [i for i in [1,2,3]]
>>> print(lista1)
[1, 2, 3]
>>> lista2 = [i for i in ("a", "b", True)]
>>> print(lista2)
['a', 'b', True]
>>> lista3 = [i for i in "Cadena de texto"]
>>> print(lista3)
['C', 'a', 'd', 'e', 'n', 'a', ' ', 'd', 'e',
 ' ', 't', 'e', 'x', 't', 'o']
```

Otro aspecto interesante es que tenemos a nuestro alcance la posibilidad de utilizar expresiones para determinar el valor de cada uno de los elementos de la lista, no estamos restringidos a que todos los elementos tengan el mismo valor. En ciertos casos puede ser interesante, y desde luego, en las preguntas del examen, lo más normal es que estos valores vengan definidos por expresiones. A continuación, veremos una simple definición de lista donde el valor de cada elemento viene determinado por el resto de dividir cada índice entre 2.

```
>>> lista = [i%2 for i in range(11)]
>>> print(lista)
[0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
```

Incluso podemos aplicar condiciones en la misma compresión. En el siguiente ejemplo mostramos como hacer una compresión que solo seleccione los índices "i" que sean pares.

```
>>> lista = [i for i in range(11) if i%2 ==0]
>>> lista
 [0, 2, 4, 6, 8, 10]
```

Podemos aumentar la versatilidad añadiendo una rama **else** a la condición. En este supuesto caso, el bloque **if-else** se situará al principio de la compresión, tal y como lo vemos a continuación, donde seleccionamos el índice par de la secuencia y en las posiciones impares escribimos la cadena "impar".

```
>>> lista = [i if i%2 == 0 else "impar" for i
                                          in range(10)]
>>> lista
[0, 'impar', 2, 'impar', 4, 'impar', 6, 'impar',
 8, 'impar']
```

Como vemos, este procedimiento para definir listas es verdaderamente versátil y nos ahorra muchísimo trabajo, y como hemos mencionado anteriormente este ahorro de tiempo es más notable cuando trabajamos con listas multidimensionales, donde todo lo que hemos visto en los ejemplos anteriores es igualmente aplicable en estos supuestos. Veamos a continuación algunos ejemplos de cómo definir listas multidimensionales utilizando las compresiones.

La sintaxis genérica para listas de dimensión 2 (para dimensiones superiores se extiende la lógica) es la que sigue.

lista = [[i for i in range(M)] for j in range(N)]

Y el ejemplo en una lista real es

```
>>> lista = [[i for i in range(11)] for j
                                        in range(11)]
>>> for fila in lista:
>>>     print(fila)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Con solo una línea hemos sido capaces de definir una “matriz” 10 x 10.

Podemos utilizar en este caso toda la potencia de los condicionales para construir las matrices que nos interese. En el siguiente ejemplo vemos como utilizamos las compresiones junto con el bloque if-else para establecer a 0 los elementos de la diagonal principal (aquellos cuyos índices cumplen la condición de que i==j) de la matriz y en el resto de posiciones escribimos los índices “i” y “j” a los que cada una corresponde.

```
>>> lista = [[0 if i==j else (i,j) for i
                   in range(6)] for j in range(6)]
>>> for fila in lista:
>>>     print(fila)

 [0, (1, 0), (2, 0), (3, 0), (4, 0), (5, 0)]
 [(0, 1), 0, (2, 1), (3, 1), (4, 1), (5, 1)]
 [(0, 2), (1, 2), 0, (3, 2), (4, 2), (5, 2)]
 [(0, 3), (1, 3), (2, 3), 0, (4, 3), (5, 3)]
 [(0, 4), (1, 4), (2, 4), (3, 4), 0, (5, 4)]
 [(0, 5), (1, 5), (2, 5), (3, 5), (4, 5), 0]
```

Con este último ejemplo, damos por concluido el apartado sobre las listas multidimensionales y las compresiones. No requiere mucho esfuerzo entender su funcionamiento y desde luego que podemos esperar preguntas con estos contenidos en el examen del PCAP.

3.3. Algunas operaciones útiles con listas

Para terminar esta lección vamos a ver algunas funciones útiles con listas y que suelen aparecer de manera recurrente en el examen del PCAP.

La primera de estas funciones son los operadores "**in**" y "**not in**". Estos operadores son capaces de revisar la lista para **verificar si un valor específico está almacenado dentro de la lista o no**.

elemento in Lista

elemento not in Lista

El primero de estos operadores "in" verifica si un elemento dato está actualmente almacenado en algún lugar de la lista y si es así devuelve True. Por otra parte, el operador "not in" comprueba si un elemento dado está ausente en una lista y en ese caso devuelve True. Para entender mejor cómo funcionan estos operadores veamos el siguiente ejemplo:

```
1.      >>> lista = ["Juan", "Lola", "Pepe", "Luisa"]
2.      >>> print("Juan" in lista)
3.      True
4.      >>> print("Tomás" in lista)
5.      False
6.      >>> print("Juan" not in lista)
7.      False
8.      >>> print("Tomás" not in lista)
9.      True
10.
```

Indicar que estos operadores también se pueden utilizar en cadenas, tuplas y diccionarios. En el caso de los diccionarios lo que comprobará si está o no en el diccionario es la clave.

Por último, vamos a ver los métodos de **ordenación de listas** que ofrece python. El primero de estos métodos es el método sort() que las ordena lo más rápido posible. Veamos un ejemplo para ver cómo se utiliza:

```
>>> lista = [3, 5, 1, 0]
>>> lista.sort()
>>> print(lista)
[0, 1, 3, 5]
```

Como se puede observar la lista se queda ordenada en orden ascendente. Si deseáramos ordenarla en orden descendente, el método **sort()** tiene un parámetro opcional llamado reverse, que por defecto es False, y que pasándole el valor True ordenará la lista en orden descendente. Este parámetro hay que pasárselo siempre por palabra clave.

```
>>> lista = [3, 5, 1, 0]
>>> lista.sort(reverse=True)
>>> print(lista)
[5, 3, 1, 0]
```

Python también proporciona una función llamada **sorted()** que crea una nueva lista ordenada a partir de la dada. Es importante conocer que este método no modifica el orden de los elementos de la lista dada.

```
>>> lista = [3, 5, 1, 0]
>>> lista1 = sorted(lista)
>>> print(lista1)
[5, 3, 1, 0]
>>> print(lista)
[3, 5, 1, 0]
```

Para terminar, python también proporciona un métdodo de lista llamado **reverse()** que se puede utilizar para **invertir la lista**. Veámoslo mejor con un ejemplo:

```
>>> lista = [3, 5, 1, 0]
>>> lista.reverse()
>>> print(lista)
[0, 1, 5, 3]
```

4. PUNTOS CLAVE

Finalmente, resumiremos los principales aspectos de los contenidos repasados en esta lección en los siguientes puntos:

- Hemos estudiado las principales características de los tipos de datos de cadena en Python. Entre otras, hemos observado la propiedad de inmutabilidad y la hemos comprobado con diferentes ejemplos.
- Hemos repasado los principales métodos y funciones utilizados para trabajar con los objetos de tipo cadena.
- Hemos visto la definición de listas multidimensionales en Python.
- Hemos visto las principales maneras de definir listas multidimensionales.
- Hemos explicado la manera más simple y compacta de definir listas mediante las compresiones de listas, de las cuales hemos visto ejemplos aplicadas a listas mono dimensionales y bidimensionales.
- Hemos visto como incluir bloques **if-else** dentro de las compresiones de listas pudiendo aprovechar así todo el potencial que estos ponen a nuestro alcance.
- Hemos visto los operadores **in** y **not in** que permiten conocer si un elemento dado existe en la lista o no de manera rápida y sencilla.
- Hemos finalizado explicando los principales métodos y funciones que proporciona Python para ordenar listas.